

RL-TR-96-200
Final Technical Report
December 1996



TESTING TECHNIQUES FOR PARALLEL SOFTWARE (TTPS)

Optimization Technology, Inc.

R.C. Cox

DTIC QUALITY INSPECTED 3

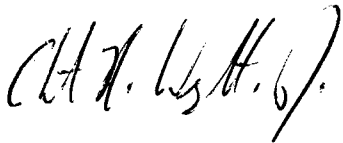
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19970218 029

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-200 has been reviewed and is approved for publication.



APPROVED:

CHESTER A. WRIGHT, JR., Capt, USAF
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO
Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE December 1996		3. REPORT TYPE AND DATES COVERED Final Apr 92 - Apr 95	
4. TITLE AND SUBTITLE TESTING TECHNIQUES FOR PARALLEL SOFTWARE (TTPS)				5. FUNDING NUMBERS C - F30602-92-C-0089 PE - 63728F PR - 2527 TA - 03 WU - 28	
6. AUTHOR(S) R.C. Cox					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Optimization Technology, Inc. 125 West Park Loop, Suite 201 Huntsvilles AL 35806				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3CB 525 Brooks Road Griffiss AFB NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-96-200	
11. SUPPLEMENTARY NOTES RL Project Engineer: Chester A. Wright, Jr., Captain, USAF/C3CB/(315) 330-4064					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The Testing Techniques for Parallel Software (TTPS) project was conducted by Optimization Technology, Inc. for Rome Laboratory. The purpose was to investigate and implement candidate parallel software testing techniques to improve the quality and reliability of parallel software through the application of automated software testing tools and techniques. Although software testing tools are beginning to mature for the sequential software development environment, the availability of software quality and test tools for the parallel environment continues to be poor. The TTPS project focused on the incorporation of candidate techniques into an operational demonstration tool. The TTPS tool delivers parallel software testing techniques, automated documentation, and metrics integrated into a high quality software quality tool ready for real-world testing application.					
14. SUBJECT TERMS Software testing, parallel software testing, software quality, PVM, statis analysis, dynamic analysis, animation, regres- sion test, reverse engineering.				15. NUMBER OF PAGES 44	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

TABLE OF CONTENTS

1.0	Introduction.....	1
1.1	Project Overview	1
1.2	Executive Summary	3
1.2.1	TTPS Objectives	3
1.2.2	Survey	4
1.2.3	Summary of Approach.....	5
1.2.4	TTPS Environment	6
1.3	Referenced Documents	7
1.4	Terms/Abbreviations	9
1.5	Document Overview	12
2.0	TTPS Capabilities	13
2.1	Static Analysis	14
2.2	Dynamic Analysis	15
2.3	Animation	16
2.4	Reverse Engineering	17
2.5	Regression Testing.....	18
3.0	Background.....	20
3.1	Parallel Software Testing Problems.....	20
3.1.1	Inter-process Coordination	20
3.1.2	Experiment Observation	21
4.0	Future Opportunities	22
5.0	Conclusion	24

1.0 Introduction

This document is the Final Report for the Testing Techniques for Parallel Software (TTPS) project. The effort was conducted by Optimization Technology Inc. under the direction of Rome Laboratories under Prime Contract PR No. C-2-2619. The project was conducted on a three year contract spanning the time from September 1993 to September 1995.

The purpose of the TTPS effort was to investigate and implement candidate parallel software testing techniques to improve the quality and reliability of parallel software through the application of automated software testing tools and techniques. Software testing tools are beginning to mature for the sequential software development environment. However, the availability of software quality and test tools for the parallel environment continues to be poor and their use virtually nonexistent.

The focus of this research was not merely the definition and documentation of theoretical techniques, but the incorporation of these candidate techniques into a working demonstration tool. The TTPS tool delivers parallel software testing techniques, automated documentation, and metrics integrated into a high quality software quality tool ready for real-world testing application.

As a consequence of the wide variety of parallel languages in use in the parallel software world, a crucial part of the TTPS effort has been the extension of OTI's multilingual technology to address parallel processing languages. The multilingual test and analysis engine forms the basis for OTI's extension of TTPS capabilities to include a wide range of target languages.

1.1 Project Overview

The TTPS research objectives consisted of a collection of complementary objectives and goals. Some of the major objectives advanced under this effort include the development of techniques to maximize parallel software testing flexibility, minimize turnaround time for test case execution and analysis, minimize experiment perturbation due to observation of behavior, provide insight into sources of non-determinism in Software Under Test (SUT) behavior, automatically detect sources of non-deterministic behavior, and detect or facilitate detection of deadlock conditions or potential deadlocks in the Software Under Test.

An overarching objective pursued by the TTPS project has been to provide the maximum degree possible of parallel software testing support in the near-term, based on OTI's multilingual test engine. The intention has been to use proven technology to get to a robust tool incorporating the new parallel software testing techniques. However, the research also addresses the more risky, high payoff candidate approaches for the longer-term incremental development/integration of additional software test and automated software quality support.

The primary accomplishment of this effort consisted of the successive builds of the parallel software quality tool called TTPS. The composite set of TTPS features are represented in the last build, Build 3, which provides a commercial grade software testing/quality tool operating in a Unix (i.e. SunOS) X Window System Motif environment. The major features supplied in this build, as is further described in Chapter 2 of this report, include the following elements:

- Automated Design Documentation Generation

- Flow Chart
- Call Tree
- Interface Diagram
- Data Flow Report (Program/File Level)
- Charts Called Report (Program/File Level)¹
- Calling Charts Report (Program/File Level)
- Multilingual approach extended to parallel structures
 - PVM/C Language Translator (available now)
 - PVM/Fortran Language Translator (expansion option)
 - MPI Language Translators (expansion option)
 - Other Language Translators (expansion option)
- Software Metrics
 - McCabe's Cyclomatic Complexity
 - Identifier Cross Reference
 - Identifier Usage Anomalies
 - Halstead's Metrics
 - Lines of Code/Comments
 - Nesting Level
- Test Coverage Analysis in Parallel Environment
 - Call Tree Coverage (Individual/Cumulative Coverage)
 - Statement Coverage (Individual/Cumulative Coverage)
 - Decision/Condition Coverage (Individual/Cumulative Coverage)
 - File Coverage Summary (Individual/Cumulative Coverage)
 - Program Coverage Summary (Individual/Cumulative Coverage)
- Visualization of execution behavior
 - Message Passing Animation
 - Control Flow Animation
 - Resource Utilization Animation
 - Resource Peak Utilization Animation
 - Resource Percent Utilization Animation
 - Host Communication Animation
- Regression Testing Support

1. Chart - under the multilingual paradigm employed by TTPS, CHART is the general term used to make reference to a subroutine/function/procedure from the originating source language.

- Graphical Differencing
- Minimized impact of test observation
- Maximized user flexibility for test observation

Other TTPS objectives focused on some of the key parallel software testing problems as identified in the survey task. For example, a major objective was to pursue techniques that eliminate or mitigate the effects of non-deterministic test case behavior in the parallel environment. As a consequence of the existence of race-conditions in the SUT, tools are needed to identify the sources of such race-conditions.

Since the run-time behavior of the software may in some cases vary significantly with small changes in timing, the act of observing the behavior of the SUT using an intrusive software tool has the potential to impact the observed behavior. Thus, a major objective must be to develop tools and techniques that minimize the impact of making the critical observations necessary to obtain the required degree of insight into the software performance.

The configuration and execution of a test case in the parallel environment may require significant resources (e.g., man hours, cpu capacity, storage capacity, etc.). Thus, the ability to rapidly configure and execute test cases with automated assistance followed by automated support for analyzing the experiment behavior is essential. Therefore, throughout the development effort, another primary objective of this research has been to maximize the user's ability and flexibility in configuring the SUT for each experiment.

Although the computing industry as a whole has attained some measure of maturity, it must be recognized that the entire parallel computing domain remains in a state of flux with respect to both architectures and languages. Therefore, an equally important objective of this research was to concentrate research and development effort on techniques that are more likely to enjoy more universal applicability to existing and future parallel environments.

1.2 Executive Summary

1.2.1 TTPS Objectives

Software reliability is a critical requirement for C³I systems and if this requirement is to be met in an efficient and cost-effective manner, automation of activities throughout the software life-cycle must be achieved. It is widely recognized that testing parallel software is much more complicated than testing a comparable amount of sequential software. Factors contributing to this complexity include the following issues:

- Non-Deterministic Execution
- Difficulty monitoring behavior
- Complex behavior
- Complications of trace data management
- Lack of automated static and dynamic analysis

Parallel software is generally susceptible to non-deterministic execution as a consequence of the existence of race conditions within the parallel software design. For example, a race condition exists if the behavior of a process C depends on the relative ordering of events within process A in comparison to process B.

The objectives pursued by the TTPS project have been to provide the maximum degree possible of parallel software testing support in the near-term based on OTI's multilingual test engine while also identifying candidate techniques for longer-term incremental development/integration.

TTPS objectives focused on the key parallel software development problems as identified in the survey task. For example, one of the primary objectives was to pursue techniques that eliminate or mitigate the effects of non-deterministic test case behavior in the parallel environment. As a consequence of the existence of race-conditions in the Software Under Test (SUT), tools are needed to identify the sources of such race-conditions.

The execution time behavior of the software may in some cases vary significantly, the act of observing the behavior of the SUT using an intrusive software tool may modify the observed behavior. Thus, a major objective must be to develop tools and techniques that minimize the impact of making the critical observations necessary to obtain the required degree of insight into the software performance.

The configuration and execution of an experiment in the parallel environment may require significant resources (e.g., man hours, cpu capacity, storage capacity, etc.), the ability to rapidly configure and execute test cases with automated assistance is also important. Thus, another primary objective of this research has been to maximize the user's ability and flexibility in configuring the SUT for each experiment.

Since the entire parallel computing domain is constantly changing with respect to both architectures and languages, an objective of this research was to concentrate the effort on techniques that are more likely to enjoy more universal applicability to existing and future parallel environments.

1.2.2 Survey

As part of Rome Laboratory's initiative to advance the state-of-the art for the development and analysis of parallel systems, OTI initially performed a survey to identify the most critical problems faced by testers of parallel software in the C³I domain.

One of the near-term goals of this survey was to assess the degree to which current testing techniques and methods address the problems, such that the requirements for a prototype testing tool could be defined. In support of the TTPS far-term goals, much effort has been directed to the identification and analysis of alternative techniques that are likely to be suitable for future integration. TTPS results include recommendations for the incremental development of a comprehensive testing environment built around the multilingual test engine derived from OTI's existing technology.

The complexity of the problems documented in the survey report indicated that current testing technology for parallel software is still fairly immature, largely as a consequence of the diversity

of parallel languages and parallel environments. The TTPS effort has made significant achievements toward a comprehensive parallel software testing environment.

1.2.3 Summary of Approach

The TTPS project was initiated with the realization that there are a wide variety of problems in the parallel processing domain and a similarly broad range of potential solution approaches for those problems. Furthermore, OTI recognized the need to identify a wide range of potential solution approaches for each problem prior to embarking on the intricate steps of software tool development, and thus OTI's technical approach included an intensive survey of the latest parallel software testing techniques. The raw data from the survey was then carefully analyzed and documented in the survey report. In conjunction with the government, the candidate list of techniques was ordered into a prioritized list, from which several techniques were selected for further development within the timespan of the TTPS project. The development and demonstration of these techniques was then pursued in a sequence of three successive builds, each adding to the functionality of the preceding build.

The general TTPS project approach may be summarized as:

- 1) Identify and document a range of alternative techniques to address parallel software problems;
- 2) Prioritize the candidate techniques for further development during the TTPS project and for future development;
- 3) Develop high priority, high benefit techniques in three successive builds

This approach was designed to take advantage of existing off-the-shelf technology to minimize schedule risk and minimize cost by leveraging the multilingual approach demonstrated by SADCA and METAsoft tools. In addition, the approach was taken with the recognition that there would be too many alternative techniques to fully develop them all during the timespan of this one project, thus the approach divided the candidate techniques into a collection of near-term techniques for immediate TTPS development and a collection of far-term techniques for potential future TTPS integration. The selected near-term techniques were implemented in the TTPS demonstration vehicle based on a modified version of the multilingual test engine from METAsoft.

The major activities of the TTPS project included:

- Parallel Software Testing Techniques Survey
- Software Requirements Definition
- Preliminary Design
- Build 1 Development
- Build 2 Design
- Build 2 Development
- Build 3 Design
- Build 3 Development

The survey activities plus software requirements definition and preliminary design were all performed during the initial project phase and the results were presented and discussed at the Preliminary Design Review. During the second phase of development, the TTPS team worked on the construction of Build #1 and the definition of detailed design elements for Build #2. These results were presented and discussed at CDR #1. During the third development phase, the TTPS team concentrated on the incorporation of PVM/C support for Build #2 and also worked on the definition of design elements for the third and final build. During the last phase, development focused on the incorporation of the remaining test case animation features for PVM/C.

1.2.4 TTPS Environment

As illustrated in the figure below, the TTPS tool was developed to operate on the "TTPS HOST" platform while the Software Under Test (SUT) may execute in any PVM/C target environment. Although the logistics of processing trace data with the TTPS tool is simplified by the availability of a network connection between the TTPS HOST and the target execution environment, such a network connection is not strictly required.

The TTPS HOST is recommended to include the following hardware and software components: 1) Sun SPARC, 2) SunOS 4.1.3, 3) X11R5, 4) Motif 1.2, 5) PVM/C 3.3.7, and 6) a shared file system accessible via NFS.

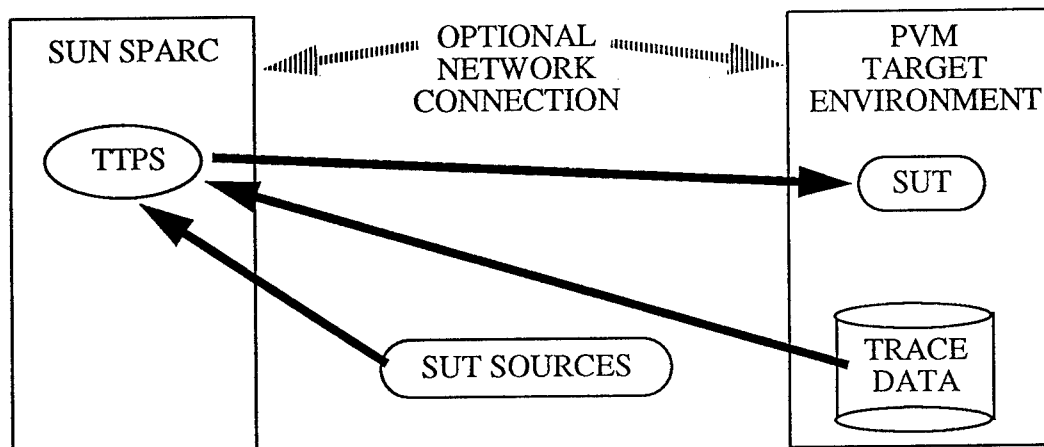
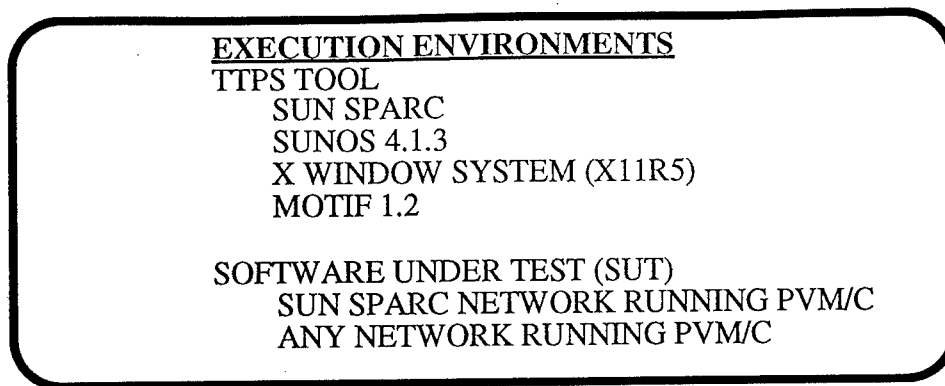


Figure 1.2.4-1 Execution Environments

Based on extensive OTI experience in applying SADCA and METAsoft tools on various software projects, the TPPS tool can certainly operate without the availability of the direct network connection to the target execution environment. In such cases, the substitute data communication mechanism is accomplished by downloading trace data sets onto a portable media such as magnetic tape and then physically transporting the media to the TPPS HOST machine for subsequent loading of trace data sets into the appropriate project database.

1.3 Referenced Documents

The following provides a summary listing of related TPPS documentation.

- TPPS Software Requirements Document
- TPPS Preliminary Design Document
- TPPS Detailed Design Document (published as multiple volumes)

- TTPS Software User Manual
- PVM User Guide

The table below shows the complete list of relevant CSC design documents for the TTPS CSCI.

Table 1.3-1 TTPS CSC Detailed Design Document References

CSC	ID	CSC Detailed Design Document
1	UI	User Interface Detailed Design
2.1	LTC	Language Translator PVM/C Detailed Design
3	LIB	Librarian Detailed Design
4	SA	Static Analyzer Detailed Design
5	TCP	Trace Coverage Processor Detailed Design
6	TRG	Text Report Generator Detailed Design
7	FC	Flow Charter/Plotter Detailed Design
8	CC	Chart Compression Detailed Design
9	CDA	Chart Difference Analyzer Detailed Design
10	ANI	Animator Detailed Design
11	TDA	Trace Data Analyzer Detailed Design
12	AS	Utilities Detailed Design

In order to perform the research described in this report, OTI reviewed numerous sources of information. Listed below are some of the books, papers, conferences, etc., that proved especially useful to OTI during the generation of this report.

- Carver, Richard and Tai, K.C. (1989); "Deterministic Execution Testing of Concurrent Ada Programs", pp. 528-544; 1989 ACM.
- Cmelik, Robert F. (et al) (March 1989); "Experience with Multiple Processor Versions of Concurrent C", pp. 335-344, vol. 15, no. 3; IEEE Transactions on Software Engineering.
- Cosnard, Michel (ed., et al) (1988); Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing; Elsevier Science Publishing Company, Inc.; New York, NY.

- Gehani, Narain and McGettrick, Andrew D. (1988); Concurrent Programming; Addison-Wesley Publishing Company; New York, NY.
- Heath, Michael T. and Finger, Jennifer Etheridge (1993); ParaGraph: A Tool for Visualizing Performance of Parallel Programs (User's Guide); Obtained through anonymous ftp from nc-sa.uiuc.edu.
- Hord, Michael R. (1993); Parallel Supercomputing in MIMD Architectures; CRC Press Inc.; Boca Raton, FL.
- MirenKov, N. N. (ed.) (1991); Proceedings of the International Conference on Parallel Computing Technologies; World Scientific Press; Singapore.
- Pancake, Cherri M. (1992); "Interview/Technical Interchange Meeting (TIM)".
- Pancake, Cherri M. (1992); "Multithreaded Languages for Scientific and Technical Computing", Technical Report CTC92TR103, Cornell Theory Center. (To Appear In Proceedings of the IEEE, January 1993).
- Phillips, T. M. (1992); "Interview/Technical Interchange Meeting (TIM)".
- Reilly, Matthew H. (1990); A Performance Monitor for Parallel Programs; Academic Press, Inc.; New York, NY.
- Shatz, Sol M. (1991); "Concurrent Software Analysis", Video Tape Instruction (2 Tapes), University of Illinois at Chicago.
- Simmons, Margaret (ed., et al) (1989); Instrumentation for Future Parallel Computing Systems; ACM Press; New York, NY.
- Solomond, Dr. John P. (Director, Ada Joint Program Office) (September 1992); "Ada Information Clearinghouse Newsletter", vol. X, no. 3; Ada Information Clearinghouse.
- Trew, Arthur and Wilson Greg (Eds.) (1991); Past, Present, Parallel: A Survey of Available Parallel Computing Systems; Springer-Verlag; Berlin, Heidelberg.
- Wilson, Gregory V. (1992); "Practical Parallel Programming"; Obtained through anonymous ftp from epcc.ed.ac.uk.

1.4 Terms/Abbreviations

This section provides definitions for some of the terms and/or abbreviations used throughout this document.

- **C³I** - Command, Control, Communication and Intelligence.
- **Coarse grain parallelism** - having a few large parallel processes requiring a relatively small amount of inter-processor communication.
- **Complete effectiveness** - technique addresses associated problem under all conditions (used to classify potential testing technique).
- **Concurrent computing** - the technology involving medium to coarse grain independent processes executing simultaneously with other similar processes.

- **Control parallel paradigm** - the application of multiple instruction streams across multiple processing elements whose execution is asynchronous between synchronization points.
- **Data parallel paradigm** - the strictly synchronous application of a single instruction stream across multiple homogeneous processing elements.
- **Data parallelism** - type of processing where the same operation or program instruction can be executed over a large array of data.
- **Dataflow paradigm** - the single assignment of variables whose computation is driven by the availability of operands.
- **Distributed memory** - storage space that is physically allocated to several processing elements. Note that a distributed memory architecture can emulate that of a shared memory architecture through the use of a single address space.
- **Fine grain parallelism** - having many small parallel processes requiring a relatively large amount of inter-processor communication.
- **Functional parallelism** - type of processing where different program instructions are executed on different processors.
- **High level of automation** - technique is automatable for all conditions, commercial grade (used to classify potential testing technique).
- **High performance computing** - the technology involving fine grain parallel segments of code executing on highly or massively parallel machines.
- **Highly parallel computers** - machines containing tens to hundreds of processing elements.
- **Internet** - the network of all globally interconnected TCP/IP networks.
- **Language class** - group of languages which implement the same type of parallelism: data, functional, or both.
- **Load balancing** - process of distributing a program among several available processors in order to achieve increased performance.
- **Low level of automation** - technique is automatable for only a few conditions, university grade (used to classify potential testing technique).
- **MIMD (Multiple Instruction Multiple Data)** - an architecture in which many instruction streams operate in parallel on multiple data sets; an architecture in which heterogeneous processes execute at different rates.
- **MISD (Multiple Instruction Single Data)** - an architecture in which many instruction streams operate in parallel on the same data set.
- **Massively parallel computers** - machines containing hundreds to thousands of processing elements.
- **Medium grain parallelism** - having several moderately-sized parallel processes requiring an average amount of inter-processor communication.
- **Message passing paradigm** - the explicit communication of element values between single or multiple instruction streams executing asynchronously between synchronization points.

- **Minimal criticality**- causes minor complications during development and testing, rarely results in failure to satisfy requirements (used to classify parallel technology problem).
- **Minimal effectiveness** - technique addresses problem under only a few conditions (used to classify potential testing technique).
- **Moderate criticality** - causes some complications during development and testing, can result in failure to satisfy requirements (used to classify parallel technology problem).
- **Moderate level of automation** - technique is automatable for most conditions, prototype grade (used to classify potential testing technique).
- **Moderately parallel computers** - machines containing a few (approximately 8 to 64) processing elements.
- **Modestly parallel computers** - machines containing a minimal number of (approximately 8 or fewer) processing elements.
- **Multicomputer** - a system in which many processors execute separate instruction streams and have access to private memories.
- **Multiprocessor** - a computer in which many processors execute separate instruction streams and share a common memory.
- **Multitasking** - the time-slice execution of many processes on a single processor.
- **Newsgroup** - any set of articles tagged with one or more universally-recognized labels.
- **Non-determinism** - the inability to achieve repeatable results from executions of a program using the same input; random ordering of events caused by physical phenomena within a system.
- **None (level of automation)** - technique is not automatable, manual grade (used to classify potential testing technique).
- **Parallel code** - distinct sets of data and instructions defining processes which execute (or appear to execute) simultaneously; these processes are sometimes identified and extracted from a single code segment as a form of optimization
- **Parallel computing** - the technology involving the collection of high performance computing and concurrent computing.
- **Partial effectiveness** - technique addresses problem under several conditions (used to classify potential testing technique).
- **Probe effect** - change in a program's behavior due to instrumentation.
- **Processing element** - a computing element that may be the collection of one or many processors and memories.
- **SIMD** (Single Instruction Multiple Data) - an architecture in which a single instruction stream operates in parallel on multiple data sets.
- **SISD** (Single Instruction Single Data) - an architecture in which a single instruction stream operates serially on a single data set.

- **SPMD** (Single Program Multiple Data) - an architecture in which many instances of a single process operate independently on multiple data sets (often viewed as an extension of SIMD or a restriction of MIMD processing).
- **Severe criticality** - causes significant complications during development and testing, often results in failure to satisfy requirements (used to classify parallel technology problem).
- **Shared memory** - a memory that appears to the user to be contained in a single address space which can be accessed by any process in a multiprocessor machine. Such a memory can actually be one physical memory or a collection of several physical memories.
- **TTPS** - Testing Techniques for Parallel Software
- **Usenet** - a set of machines that exchange articles tagged with one or more universally recognized labels, called newsgroups; the locations of these machines include government agencies, large universities, high schools, businesses and even home computers.
- **Von Neumann architecture** - computer architecture in which only one instruction executes on one data item at a time.

1.5 Document Overview

Chapter 1 provides an introductory description of the TTPS project and a summary of the technical results. Chapter 2 provides a summary description of the capabilities currently available in the TTPS tool. Chapter 3 presents a brief overview of parallel software technology issues that must be addressed by TTPS. Chapter 4 describes some of the immediate options for further TTPS development and enhancement. Chapter 5 summarizes the conclusions to be drawn from the TTPS research and development effort. The appendix presents an overview of the candidate techniques identified and documented as a result of the survey task.

2.0 TTPS Capabilities

TTPS brings a broad range of software testing capabilities to the parallel processing domain that have previously been available only to sequential software. TTPS provides five major categories of functionality:

1. Static Analysis
2. Dynamic Analysis
3. Animation
4. Regression Testing
5. Reverse Engineering.

Each of these areas and their benefits to parallel software development and testing is further described in the following subsections. TTPS capabilities can be useful in a broad range of software development life-cycle activities including: programming, testing, quality assurance, IV&V, and project management. TTPS provides capabilities to automate the analysis of source code with the collection of a variety of software quality measures including software metrics and test case coverage.

OTI has been successfully developing and applying automated software testing technology since 1988 when the original Static and Dynamic Code Analyzer (SADCA) tools were first fielded and applied to an ongoing government program. The original SADCA implementation targeted the CMS-2 language (a specialized language for programming embedded computing systems targeting primarily the AN/AYK-14 CPU's). Principle features of this version included static analysis of variable usage, dynamic coverage analysis measurement and reporting, plus code metrics measurement and reporting. The success of CMS-2 SADCA application on the Exo-Atmospheric Reentry Interceptor System (ERIS) project immediately led to the development of a comparable tool targeting the Modula-2 language. The Modula-2 SADCA was an equally successful development and was used for the analysis of all of the launch operations software for the ERIS project including the Mission Launch Control Processor (MLCP) software and the Launcher Operations (LO) software. SADCA successes have also been instrumental in carrying the technology over into the Brilliant Pebbles and the Payload Launch Vehicle programs where SADCA has delivered additional benefit.

The CMS-2 and Modula-2 SADCA tools led to the formation of the multilingual test engine concept based on the notion of a common language encompassing the structural properties of each individual language. The rationale for this concept was the ability to process multiple source languages down to the same database representation, thus making it inexpensive to incorporate new source languages since the analysis tools and report generator tools on the back-end can be used without modification. Thus, the definition of the Meta-language has become the foundation of subsequent SADCA development and in 1990 became the core data representation used in the ARC SADCA multilingual tool. ARC SADCA was developed for the ARMY's Advanced Research Center (ARC) located in Huntsville, Alabama and was developed with support for Fortran, Ada, and several dialects of C including K&R, ANSI C, and DEC C.

ARC SADCA technology has been successfully applied to a variety of projects including: High Endo-Atmospheric Defense Interceptor (HEDI), Joint Theater Level System/Joint Warfare Center (JLTS/JWC), Kinetic Energy Weapon Digital Emulation Center (KDEC), Ground Based Radar (GBR), UK Testbed, Radar Imaging Defense System (RAIDES), etc. The benefits of the multi-ligal test engine have been demonstrated repeatedly.

Although the SADCA/METAssoft architecture was designed to support software testing processes for sequential software, the opportunity for supporting parallel software testing was clear. Thus, the TTPS project was initiated with the intention of using the SADCA/METAssoft architecture as the point of departure for the development of a highly functional demonstration vehicle for testing parallel software. This effort has been quite successful in revising METAssoft structures for the new TTPS requirements.

2.1 Static Analysis

The static analysis capabilities can be employed from the earliest stages of software development to identify errors/potential errors and suspect routines early in the development. Early detection typically yields a larger benefit to a given project since it reduces or eliminates effort expended later in the project timeline.

Static analysis services are based on the automated examination of the structural characteristics of the code. TTPS static analysis services provides a variety of features:

- automated complexity measurement in support of reliability, testability, and maintainability
- early identification of potential problem areas requiring greater attention
- positive identification of areas with inadequate testing
- quantification of testing in support of quality management
- detection of error conditions in data usage
- identification of departures from approved coding standards

TTPS supports the automated collection and reporting of a variety of metrics in support of management control over the software development and software testing processes. Specific metrics provided by TTPS include the following:

- cyclomatic complexity
- halstead's software science
- source lines of code
- comment percentages
- nesting level

TTPS error detection capabilities can be instrumental in the elimination of software defects prior to the first execution of a test case, thus saving time and money from the project budget. Some errors detectable by TTPS static analysis include the following set of conditions: 1) statically infinite

loops; 2) inconsistencies with identifier declaration, initialization and use; and 3) unused or unreachable code segments.

2.2 Dynamic Analysis

The dynamic analysis capabilities can be employed from the time in the life-cycle that executable software becomes available. The focus of dynamic analysis activities is to execute test cases on the SUT, observe the run-time behavior of the software, and perform post experiment analysis of the observations to quantify specific characteristics of the experiment. A characteristic of particular importance in evaluating the benefit of specific test cases is the degree of coverage achieved by one or more tests. A major objective of the structural testing approach is to ensure that the code has been exercised to the required degree. Typical goals for structural testing include the following:

- Executed each statement in the SUT at least once.
- Execute each decision outcome (TRUE/FALSE) for each decision statement at least once.
- Execute each condition outcome (TRUE/FALSE) within each decision expression of a decision statement at least once.

The tool user is free to choose the degree of test coverage desired. The TTPS tool will support the measurement and reporting of the degree of test coverage achieved.

In summary, specific TTPS features providing distinct programmatic benefits include the following:

- automated support for measurement and reporting of test coverage
- objective quantification of testing effectiveness
- support for objective evaluation of code quality
- provide guidance in the test case development process
- provide for increased confidence in code quality
- reduction of testing redundancies and insufficiencies (identifies "Holes" in test suite)

Quantification of test coverage is provided by TTPS in terms of detailed coverage reports that may be presented in either color-coded graphical form on an X Windows System display or may be produced in summary text format. In either format, the resulting reports may easily be printed to a common postscript compatible printer. Specific coverage reports provided by TTPS include the following:

- call tree coverage reports
- statement coverage reports
- decision coverage reports
- condition combination coverage reports

2.3 Animation

TTPS provides test case animation features to address the problem of complex behavior of parallel software. Parallel software behavior commonly exhibits complex timing relations between the various processes comprising the parallel program. As a consequence, it is very important that these timing relations be thoroughly understood and compared with the required behavior. However, it is frequently the case that the software development environment includes little, if any, support for the objective evaluation of these timing relations. The objective of the TTPS animation services is to facilitate these evaluations through the direct measurement and reporting of execution behavior.

Animation features supported by TTPS include the following:

- visualization of test case behavior
- presentation of alternative views of complex timing relationships of parallel software
- increased confidence in code quality
- guidance in test suite development

The Build 3 animation set represents a sample of what functionality can be provided with the raw performance data that TTPS can collect and report. Multiple animation views can be displayed on the screen simultaneously, thus facilitating the time-wise correlation of various types of information presented in the graphics.

The user interface for the animation services is configured with several important options for the TTPS user's benefit. For example, VCR-like controls are provided in the Animation Executive window to control the Start and Stop times for a given animation sequence. By limiting the animation timespan, the user can effectively "zoom-in" on the events within the selected timespan.

In addition, "Process Enablement" options are provided such that the user may focus on the behavior of a smaller subset of processes. The animation process will display graphics including the events only from those processes that are currently enabled. Flexibility in the user interface also allows for alternative views to be produced from the same trace data set quickly and easily without the need to reload the data set from the database.

TTPS provides the following animation capabilities:

- message passing animation
- control flow animation
- resource utilization animation
- resource peak utilization animation
- resource percent utilization animation
- host communication animation

The message passing animation feature provides a graphical display of the interprocess message passing activities and other event times. For example, the graph illustrates the task start time, each message send event, each message receive event, each task creation event, and each task termina-

tion event. Depiction of all these events on the same global timeline allows the comparison of observed events with the expected behavior.

The control flow animation is a very detailed animation capability permitting the TTPS user to observe the sequence of events with respect to the control flow structure of the SUT. The control flow animator displays the same kind of flow chart diagrams familiar to the TTPS user from the Static and Dynamic Analysis capabilities. However, for animation, the central purpose of the flow chart is to convey to the user the location in the code that corresponds to the current event. The exact component corresponding to a given event is positively identified by color coding the component (i.e. the default color, if not customized by the user, is yellow).

The resource utilization animation graphics use the timestamped trace data to construct a diagram that indicates how CPU resources are used by the various processes of the parallel program. The diagram includes one row for each of the host CPU's existing in the user's parallel virtual machine. Within each row of the diagram, the Busy/Idle status for the CPU will be indicated for each instant in time via color coding. The default colors will use green to indicate a busy status and will use yellow to indicate the idle status. The resulting diagram indicates the time-relative CPU utilization for the entire virtual machine.

The resource peak utilization animation is a derivative of the resource utilization diagram. It exhibits a similar graphic structure, however, the information presented at each instant of time within the diagram is not simply the Busy/Idle status. Instead, the data item illustrated is the instantaneous number of processes active on the specific CPU (i.e. how many different processes currently sharing the CPU of a given node of the virtual machine).

The resource percent utilization animation is another derivative of the resource utilization diagram. It has a similar structure, but its information content is different. The timeline is uniformly divided into a sequence of time-slices. During the time interval for each time-slice, the busy/idle status for a given CPU is sampled, from which a percent utilization statistic is calculated for the specific CPU over that time interval. The percent utilization statistic is the information that is plotted within the diagram. The resulting diagram shows the relative loading of the various nodes of the virtual machine as a function of time.

The host communication animation is a special derivative of the message passing animation service. It presents an alternative view of the message passing structure with emphasis on the interhost message passing. Thus, highlighting the host-to-host communications. The diagram is a direct mapping of the process-to-process message passing structure onto the physical hosts.

2.4 Reverse Engineering

TTPS provides a variety of reverse engineering features to facilitate the parallel software testing process. This can be particularly useful in those cases where the existing design documentation is inadequate. The service provides a basis for quickly regenerating design elements from the actual code. Software testing can then proceed with this design information available.

Independent review of parallel software is facilitated by the reverse engineering capability. The software test personnel can produce succinct reports documenting the design embedded in the

SUT. In those cases where some design documentation exists, this capability supports the comparison of the design attributes of the actual code with that of the existing documentation.

In summary, the TTPS reverse engineering features include the following capabilities:

- documentation assistance for undocumented code
- design verification
- visibility into data and control flow
- comprehensive view of system
- assistance in interface verification testing

TTPS provides two general categories of reverse engineering reports: 1) code structure oriented reports and 2) data oriented reports. The structure category includes the following structured flow diagrams:

- call level
- control flow by routine with on-screen source mapping
- interface diagram

The data category includes the data oriented reports listed below:

- complete identifier cross-reference
- data flow report
- charts called outline
- calling charts outline

2.5 Regression Testing

TTPS provides significant new services to the parallel software testing domain in the form of regression testing support. The benefit of the services is geared to improve the efficiency of regression testing processes following changes to the software configuration of a parallel program.

Specific regression testing features provided in TTPS include the following:

- automated identification and documentation of version differences
- identification/recording of changes to control flow logic
- providing guidance for test suite development and optimization

An important problem that exists in software testing in general, is the need to retest a program following one or more changes to the software configuration. Any alteration of the source code may potentially require the entire suite of test cases to be executed again since the alteration may change the overall program behavior. The TTPS regression testing capability mitigates this effect by assisting in the identification of specific locations where changes have been implemented and where retesting is required. This problem is more acute in the parallel domain since changes may affect the interprocess timing relations, which may in turn lead to changes in behavior.

The comparison of different versions is accomplished using graphical differencing techniques, as opposed to the more traditional line-by-line source text comparison. The approach implemented by TTPS has the distinct advantage of ignoring changes to the source text that have no potential impact on the execution behavior. Specifically, TTPS will ignore changes to the formatting, spacing, and comments. The important differences reported by TTPS are limited to the logic structure changes and any modifications to the declaration and usage of data items. Furthermore, human interpretation of the reported changes is facilitated by formatting the difference reports in a color-coded graphical form such that the context of a report change is readily apparent from the structure of the graph. Furthermore, the color coding facilitates the identification of segments of code that need to be the focus of additional tests in regression testing as a direct consequence of a code modification. For instance, a block of code that immediately follows a block in which a change has been identified must be the subject of regression testing since its behavior may be affected by the change(s).

In summary, the TTPS regression test support provides the following types of reporting services:

- identification of changes between baseline software versions
- visual mapping of source changes to control flow effects
- control flow tracing of potential ramification of changes

3.0 Background

This section provides a brief summary of the TTPS and parallel software background sufficient to facilitate the comprehension of the TTPS project with respect to the parallel processing domain. Clearly, the complete topic is sufficiently broad to be fully addressed only with a rather large collection of books and technical papers. Instead, the following sections will provide summary background on problems encountered with parallel software and the OTI technology base supporting the TTPS development.

3.1 Parallel Software Testing Problems

Examination of the open literature describing the spectrum of parallel hardware and software architectures reveals a wide range of possibilities with respect to applicable techniques. Parallel computing has taken many shapes and forms over time and continues to evolve. Over time one can recognize the recurrence of specific problems and solution approaches. Some of the problems that seem to remain constant regardless of the specific implementation include the following:

- coordination of timing between independent processing activities
- observation of process behavior.

A variety of techniques have been proposed, developed and tested for a wide variety of environments with varying success. In general, the parallel software developer needs a range of alternatives from which to choose the most appropriate technique for his immediate problem. This calls for automated tool support to facilitate rapid construction and execution of experiments followed by quick post-experiment data analysis.

3.1.1 Inter-process Coordination

The coordination of activities between multiple processes is a fundamental problem in parallel computing. In general, the problem arises from the need to transfer information back and forth between two or more processors. The mechanisms for implementing the information sharing falls into one of these categories: 1) shared memory, 2) message passing, or 3) hybrid.

With the shared memory configuration, multiple processors are provided with direct access to a common memory device. These processors have unlimited access to the shared memory resource, and without some form of predefined means for cooperation, can easily produce invalid results as a consequence of data race conditions. For example, two processors may read the contents of a shared memory cell (MC1) in addition to accessing other data items, after which they both perform some computations using this value and then write the new result back to MC1. In the real machine the read and write events will occur as a time-ordered sequence of events, with one machine's write operation preceding the other machine's write operation such that one result is overwritten by the second Write operation. The parallel software developer must address such potential race conditions through the careful construction of specific software structures tailored to the given machine architecture. For example, process coordination has frequently been accomplished via the consistent application of software semaphores.

In contrast to the shared memory architectures, distributed architectures perform their interprocess communication and coordination via message passing. In the absence of shared memory, the only means for communications and coordination between multiple processors is via explicit messages. Messages may be passed over whatever communications media is available in the specific architecture. It should be noted that restriction to a particular type of communication media would be entirely arbitrary. The central issue of the message passing paradigm is that the information flow from process to process is composed of discrete packets of information (i.e. messages).

In the general case for distributed parallel computing environments, the compute nodes of the distributed machine may be dissimilar and may have different interconnect structures between them. For instance, some compute nodes may incorporate multiple tightly-coupled processors having shared local memory structures. In such cases, the architecture may be described as a hybrid of shared memory and message passing. In view of the current trends in computing, it seems clear that this approach is likely to continue to grow in importance.

3.1.2 Experiment Observation

In general, the observation of the behavior of the SUT requires one of two classes of data collection techniques: 1) hardware probe based, and 2) software probe based.

The hardware probe approach has both advantages and disadvantages. Its advantage is the ability to observe the behavior of the SUT without disturbing the timing relationships of the experiment. However, its principle drawback is the difficulty of implementing and paying for the required hardware support. In general, hardware vendors do not encourage the custom modification of the equipment they sale and will likely void any existing warranties. In addition, the hardware approach is generally not scalable to larger parallel computing architectures.

The software probe approach has the disadvantage of perturbing the experiments timing and potentially modifying its timing relationships. However, the approach has the following distinct advantages:

- more readily scalable to large configurations
- can be applied with a large degree of independence with respect to the hardware architecture
- can be used with a wide range of programming languages.

TTPS implements this approach since it maximizes the benefits while minimizing the disadvantages.

4.0 Future Opportunities

The TTPS effort has demonstrated the powerful support techniques for software testing/quality for parallel software. Other possibilities exist for further development of the TTPS tool's capabilities. For the sake of discussion, these possibilities are described below in terms of the immediate possibilities for capability extensions.

The TTPS multilingual architecture inherited from SADCA/METAsoft has demonstrated the feasibility of integrating new functionality at minimum additional cost. Specific new features that may be integrated include the following:

- New language translator: PVM/Fortran
- New Language Translator: MPI/C
- New Language Translator: MPI/Fortran
- New Language Translator: Other message passing languages/subroutine libraries
- Additional Animation Graphics
- Deterministic Replay of a specific test case
- Specification and automated checking of real-time constraints
- Run-time checking of real-time constraint assertions
- Post-experiment checking of real-time constraints
- Hybrid instrumentation techniques

The multilingual test engine at the core of TTPS has been designed to facilitate the integration of new languages with minimal effort. This attribute should minimize the effort required to integrate new languages such as PVM/Fortran, MPI/C, and MPI/Fortran. In addition, the PVM/C experience shows that the integration of other languages/environments based on the message passing paradigm should be tasks of comparable size. Furthermore, the extension of TTPS to support distributed multilingual environments seems technically feasible with little impact on the TTPS architecture.

Much additional work could be performed in the area of animation graphics development. TTPS provides the basic mechanisms for timestamped event collection, storage, and retrieval with several example animation capabilities in operation. With some modification of the instrumentation process, it seems feasible to extend TTPS to include support for the Deterministic Execution Replay feature.¹ In contrast to the animation feature which presents graphical replay based the contents of a static trace data file, the deterministic replay feature would permit the re-execution of the parallel program while forcing the same sequence of interprocess coordination events. The approach would require two different types of instrumentation: 1) instrumentation designed to collect the data required for replay; and 2) instrumentation designed to use the recorded event sequence to force the recurrence of the same sequence in the re-execution of the SUT.

1. Deterministic Execution Replay (Technique #12) in the Appendix

The existing TTPS structure can provide a vehicle for specification and automated checking of real-time constraints. Checking could be performed either at run-time and/or after experiment completion. The existing TTPS facilities for user-directed insertion of instruments to enable/disable timestamped event data have proven the feasibility of user-directed instruments. The real-time constraint checking feature represents an extension of the current TTPS instrumentation capability.

Although the general application of the Hardware Instrumentation Technique¹ on a large scale is not economical, there may be instances in which a hybrid approach of Hardware and Software instrumentation may be both desirable and cost-effective. In the current era of local area networks, it is entirely reasonable to expect that the parallel computing environment may include some compute nodes having special characteristics, such as hardware instrumentation. If the data collection process implemented by the hardware instrumentation produces trace information in the format recognized by the TTPS trace data analyzer, then the actual source of the trace data (i.e. whether hardware instrumentation or software instrumentation) becomes unimportant.

1. Hardware Instrumentation Technique (Technique #1) in the Appendix

5.0 Conclusion

The TTPS project has successfully demonstrated the feasibility of providing a range of advanced software testing/quality techniques in the parallel environment. TTPS provides an extensive array of capabilities in support of PVM/C software development and testing and provides an incremental growth path for the incorporation of support for many other parallel programming languages. Furthermore, as a consequence of the outgrowth of the TTPS technology from OTI's commercial METAsoft technology base, the TTPS tool provides these capabilities in a commercial grade, high quality tool available on popular, low-cost Unix workstations.

A.0 Appendix

A.1 Survey Results

The survey effort utilized a variety of sources including direct contacts, mailings, Internet mailings, and Internet searches. From these contacts, a collection of techniques was gathered and documented in the TTPS Survey report. These findings are presented in the table below followed by summary descriptions of each technique.

Table 1: Survey Report Candidate Techniques

Technique	Identification
1	Hardware Instrumentation
2	Kernel Instrumentation
3	Uncorrupted Trace Recovery
4	Advanced UTR
5	Selectable Instrumentation Levels
6	Minimization of Probe Effect (type 1)
7	Minimization of Probe Effect (type 2)
8	Identification of Synchronization Sequences
9	Graphical Execution Replay
10	Graphical Execution Differencing
11	Deadlock and Data Race Detection
12	Execution Replay
13	Variable Analysis
14	Static Analysis

A.1.1 Technique #1 - Hardware Instrumentation

Technique #1 is a hardware-based approach to the collection of information describing a program's execution behavior. It applies custom hardware to the problem of collection and reporting of execution behavior. The technique has been attempted in a variety of forms, for example [Tsai90] as mentioned in the TTPS Statement of Work describes one project's experience with this approach. In the general case, this technique requires the construction of custom hardware to physically mon-

itor the operations of each CPU and record specific pieces of information for subsequent post-experiment analysis.

The major benefit of the hardware approach is the potential to eliminate, or at least minimize, the perturbation of the experiment as a consequence of observing the experiment behavior. With dedicated hardware, it becomes feasible to observe execution behavior with no change to the normal execution timing characteristics.

The drawbacks of the hardware approach are associated with the cost of implementation. The technique requires the design and development of custom hardware for each new type of CPU to be supported. Similarly, the technique is not easily scalable to larger systems since each CPU whose activities are to be monitored requires the additional hardware support. Observing execution behavior on N nodes requires N hardware monitors plus their supporting communications and centralized data collection facilities.

A.1.2 Technique #2 - Kernel Instrumentation

Technique #2 is an alternative approach to the collection of information describing a program's execution behavior. In general, one may approach the problem of execution behavior monitoring with focus on hardware or software support. Technique #2 uses the software approach but places the software instruments within the kernel (i.e. the Operating System) itself. An example of this approach is found in DPM (Distributed Programs Monitor), which is a simple design for monitoring the execution and performance of distributed programs [Miller88].

The simplicity of DPM has resulted in the creation of a system of tools with a wide range of uses that have minimal affect on a program's performance. Monitoring is based on detecting and recording message interactions using functions that are built into a modified system kernel. The data provided by DPM is obtained by inserting software probes into the operating system at locations that process interrupts or provide support for user programs. DPM provides facilities for both post-mortem analysis and playback of trace data and real-time monitoring of programs as they run. Filtering of trace data is performed to minimize storage requirements and reduce I/O.

The most significant problems addressed by DPM are the levels of instrumentation and timing constraints (probe effect). The approach does not actually eliminate the effects of software instruments on execution behavior but it does serve to significantly reduce such effects.

A.1.3 Technique #3 - Uncorrupted Trace Recovery

This technique involves a computationally intensive post-mortem method for recovering segments of uncorrupted event traces. The recovery technique described is independent of the type of probe used to provide the event traces (within the assumptions of when the recovery technique works) and resembles control-theoretic perturbation analysis. The recovery process uses information about program structure (encoded using timed Petri-nets), together with knowledge of the individual delays caused by the isolated monitoring probes, to reliably reconstruct parts of the event trace that would have occurred if no monitoring had taken place. The production and use of timed Petri-

nets required by this technique is "straightforward, albeit tedious", and appears amenable to automation.

A.1.4 Technique #4 - Advanced UTR

This is a technique that enables segments of uncorrupted event traces to be recovered from corrupted event traces. The paper mentioned below further investigates this technique and provides more details of a (slight) generalization of the original technique.

The goal of this technique is to take the corrupted event trace information that was obtained during the monitoring process, and recover from it as much as possible of the *exact* control flow information that would have been observed had the monitoring process itself not caused any timing perturbations. As compared with [Andersland91], [Lumpp92] concentrates on distributed memory parallel computing systems that make extensive use of message passing. The techniques described in both papers are of considerable interest, since at present the only way to obtain event trace data that has not been perturbed by the monitoring process itself is to use some form of hardware monitoring. However, hardware monitoring is extremely hardware dependent and costly.

A.1.5 Technique #5 - Selectable Instrumentation Levels

This technique provides a common sense approach to providing the appropriate level of program instrumentation. Often a predefined level of instrumentation is assumed and applied to a program under test resulting in excessive trace data and intrusiveness. In order to minimize intrusiveness and unwanted trace data, the process of instrumentation must be optimized. This is accomplished by allowing the user to select from a set of predefined instrumentation levels (i.e., process communication, function calls, statement execution, etc.). In conjunction with allowing the user to specify an instrumentation level, the portions of the program to be instrumented are selectable. This allows the user to focus analysis efforts on specific areas of the program. Once selected, the necessary instruments are applied to the program under test.

The problems addressed by this technique are level of instrumentation (difficulty in monitoring parallelism), timing constraints (probe effect), level of instrumentation (trace data management), trace data collection and storage, and trace data display

A.1.6 Technique #6 - Minimization of Probe Effect (type 1)

There are many techniques for minimizing probe effect. However, two of the most notable methods are the storage of intermediate trace data in memory and limiting the points at which trace data is transferred to disk. Utilizing memory for the intermediate storage of trace data is notably faster, since memory access operations require significantly less time than disk access operations, thereby limiting the effects of instruments on program timing (probe effect). However, limited memory size restricts the amount of data that can be stored in this fashion. Therefore, periodic flushing of memory to disk is required. If the flush operation is restricted to program synchronization or user specified points, the overhead caused by disk I/O will also be minimized. The flush operation may

be aggravated in some parallel systems, since all nodes of a parallel machine may not support direct I/O to a file system.

The problems addressed by this technique are timing constraints (probe effect) and trace data collection and storage

A.1.7 Technique #7 - Minimization of Probe Effect (type 2)

Alternative techniques for minimizing probe effects are demonstrated in the IPS-2 system. First and foremost, IPS-2 is a performance measurement system for parallel and distributed systems. It is based on an abstract hierarchical model of both parallel and distributed computing. Using the levels of this hierarchical model, it can provide multiple views of the performance (and correctness) of a parallel program. By moving from one level of the model to another it is possible to increase or decrease the amount of detail provided to the tool's user. This zoom capability can greatly aid the location of problems in parallel code (and, of course, the discovery of performance bottlenecks).

In an attempt to minimize the probe effect caused by monitoring programs, IPS-2 uses several techniques: multiple levels of instrumentation, avoidance of probe routines using (or being implemented as) operating system kernel calls, combination of hardware and software probes, compression/reduction of collected data, caching of collected data, combining trace data for related events.

The multiple level hierarchy modeled by IPS-2 provides a good basis for providing the user with multiple levels of instrumentation. Without altering program source code, the user is able to specify the level of detail that probes should provide. At present (moving from the coarsest to the finest level of detail) these levels are: program level (the distributed program is seen as a black box running on the system), machine level (the program consists of multiple threads running simultaneously on multiple processors), process level (the program is represented as a collection of communicating processes in which machine boundaries can, if desired, be ignored), procedure level (each process of the program is represented as a sequentially executed chain of procedures) and primitive activity level (at which such activities as process blocking/unblocking, message send/receive, process creation/destruction and procedure entry/exit are represented). By providing the user with choices as to which level of instrumentation should be used, IPS-2 can turnoff all instruments not required by the particular level chosen, and reduce to a minimum the probe effect that results from monitoring the program.

A.1.8 Technique #8 - Identification of Synchronization Sequences

The technique described in [Taylor83] statically analyzes the synchronization structure of parallel programs. From the synchronization structure, this technique can be used to identify execution paths, potential deadlock conditions, and parallel actions within the program under analysis.

Inputs required by the algorithm employed in this technique include: a program call graph, program scope information, and an annotated flowgraph. Program call graphs provide information about the subprogram invocation structure (i.e., the units callable from within each program unit). The call graph is used by the algorithm to identify all program units capable of performing task

(process) activities (e.g., task activation, task rendezvous (synchronization) and explicit delay statements). Program scope information indicates the nesting or hierarchical structure of the program units. This information is used by the algorithm to determine the nesting of task declarations. Program flowgraphs graphically represent the flow of control within a program. Nodes of the flowgraph typically correspond to statements within the program. Since the analysis performed by this technique requires only the synchronization structure of the program, other structural information usually contained in flowgraphs is not required. Flowgraphs for this technique must contain nodes for communication calls, task activations, etc.

A.1.9 Technique #9 - Graphical Execution Replay

This technique, in a general sense, allows the graphical replay/animation of program execution histories (it does not force the actual re-execution of the executable image). Trace data generated as a result of the execution of instrumented programs is required as input for this technique. The format for the input data has been defined to be comparable to that employed by PICL, but with some improvements in efficiency.

The correct ordering of the trace data events is required in order to ensure the correctness of relevant event timing as viewed on a graphical display. Each trace data record saved during the test case execution includes a timestamp derived from the local clock to facilitate global ordering of events following completion of the test. Global ordering of events is required in order to display the event traces in a single animation display relative to a global timeline.

Animation services of various types can be constructed based on timestamped trace files. Examples include the following: processor status, percent CPU utilization, message passing structure, etc.

A.1.10 Technique #10 - Graphical Execution Differencing

This technique provides a graphical difference of program execution paths. Trace data generated as a result of the execution of instrumented programs is required as input for this technique. Trace data obtained from several executions of the program under test can be used to graphically display the paths recorded during program execution, and to highlight unique execution path segments and nondeterministic execution points.

The level of detail provided by the graphical difference depends on the input data (trace data) level of detail. For example, if only task interactions and subroutine calls were recorded by instruments during execution, this technique would reflect execution history differences at a high level. However, if the program were fully instrumented, the user would be able to view execution history differences at a low level (e.g., statement execution differences versus task interaction differences).

The problems addressed by this technique are the problems of execution visualization, replay of execution, and trace data display

A.1.11 Technique #11 - Deadlock and Data Race Detection

A technique is described ([Hariri92], [Netzer92] and [Schatz92]) which will support the analysis of parallel software so that race conditions can be detected. Although the technique is discussed in

a slightly different vein in each reference, the intent is to identify the code and execution scenarios which cause race conditions. By isolating these conditions, the cause of nondeterministic execution will also be identified.

The common thread among variations of this technique is the use of abstracted program data (i.e. graphs) to determine the cause of race conditions. Each graph is a means by which statements containing conflicting data accesses can be identified, and a determination can be made about whether these statements are executed in a well-defined sequence.

A.1.12 Technique #12 - Deterministic Execution Replay

This analysis technique uses synchronization sequences to control the replay of parallel program execution scenarios. As described in [Tai86] and [Tai91], guaranteeing deterministic program execution in this fashion involves the collection of, and subsequent use of, these sequences of thread synchronizations (i.e. SYN-sequences) to force a particular execution path(s) to be taken within a parallel program. SYN-sequences guarantee a particular execution sequence of program threads, given a predefined input. As per [Tai86] and [Tai91], the use of these sequences is defined for the Ada programming language.

A program (P) is instrumented to produce a modified version (P') of the original program. P' is then executed to generate a list of Ada task synchronization sequences. Upon subsequent executions of P', execution can be replayed given the recorded synchronization sequence as input, in addition to the original inputs. The realized execution path(s) and outputs are expected to be identical to those experienced during the first execution of P'.

The technique, as currently defined, addresses a subset of the Ada programming language. Each of the following Ada constructs is NOT included in the application of this technique: 1) selective waits with two or more accept alternatives for the same entry, 2) selective waits with two or more alternatives that may have the same delay value, 3) accesses to shared variables, 4) abort statements, and 5) statements accessing the real-time clock.

The problem addressed by this technique is the lack of deterministic execution (i.e. non-determinism) experienced by many parallel programs

A.1.13 Technique #13 - Variable Analysis

In [Taylor80] several static analysis techniques are described that permit the automatic detection of program anomalies. The types of anomalies detected include "variable usage errors" (e.g., the use of variables before they have been initialized, the multiple initialization of variables which are not used between initializations, the initialization of variables that are never used) and "scheduling/waiting errors" (e.g., waiting for a process that is not scheduled to run, waiting for a process that has already finished running). In this paper, these and other potential errors are shown to be detectable (with some restrictions) using static data flow analysis. Although these errors are of significance in themselves, they are often an indication that deeper errors are present at the design level.

In later, more general work (e.g., [Long91]), which is also discussed, these techniques are generalized and shown to be amenable to automation.

A.1.14 Technique #14 -Static Analysis

The starting point for structural testing of sequential programs is the flowgraph model of the program. This graph consists of nodes, each of which represents a statement or collection of statements, connected by edges which represent the flow of control from one block of code to the next. Nodes that possess several exiting edges represent either a branch predicate or a collection of statements with a branch predicate as the last statement.

Structural testing consists of performing various types of tests on a program's flowgraph. The most extensive testing is path testing, which consists of using suitable test cases to check that all possible paths are exercised, and that the code within them works as intended. For sequential programs, path testing can be a time consuming testing process that requires the generation and use of data that exercises all the available paths. For all but the most simple of parallel programs, path testing becomes impractical since the number of possible paths is usually very large, if not infinite.

The next most extensive testing technique is that of "all-DU-paths" (definition-use) testing, which, according to [Taylor92], "consists of checking every definition-clear subpath" (in the flowgraph) "from every definition to all the successor nodes of each use reached by that definition that is a simple cycle or [is] cycle-free". This technique was shown in [Weyuker90] to be both practical and worthwhile for well-written code, but is not the focus of [Taylor92].

Some of the least effective, but perhaps most applicable, testing methods are those of branch testing and statement testing, which have the advantage that even for parallel programs, the required test set is finite. Since these methods are not completely effective at detecting errors, they should be used only as a part of a more comprehensive testing scheme.

A.2 Prioritization of Techniques

In addition to gathering an extensive collection of information describing the various candidate techniques described earlier, the first phase of the TTPS effort also required the prioritization of the candidate techniques. The figure below illustrates the prioritization of problems that guided the development of the TTPS tool and the techniques supported by the tool.

The prioritization of candidate techniques relative to key characteristics (i.e. effectiveness and level of automation) identifies those techniques that are most likely to address the problems critical to the C³I parallel software application domain. The technique evaluations and prioritizations were used in selecting the techniques pursued by OTI in the near term and also serve to guide the choices for long term development.

The ordering of the following list of techniques was based on each technique's effectiveness at addressing one or more parallel testing problems. However, in order to be fair, two factors (frequency of use and criticality of problem addressed) were used to properly weight the decisions necessary

to achieve the final ordering. The first step required to perform the ordering was to identify an "effectiveness weight (EW)" for the effectiveness classifications of **highly** effective, **partially** effective and **minimally** effective. These classifications were used for technique evaluations in the previous sections. The classifications were weighted as follows:

highly effective:3 units
moderately effective:2 units
minimally effective:1 unit

Secondly, it was necessary to adjust the EW of each technique based on the criticality of the problem addressed. Given the ordering/prioritization of problems, the following "criticality factors (CF)" were defined:

Nondeterminism (3.4): 1.0
 Difficulty in Monitoring Parallelism (3.3): 0.8
 Complex Behavior (3.2): 0.6
 Trace Data Management (3.5): 0.4
 Common Static and Dynamic Analysis Issues (3.1): 0.2

Using the terms defined above the following formula was used to determine the relative effectiveness of each technique:

$$E(T_n) = EW_{T_n}(P3.4) * CF_{P3.4} + EW_{T_n}(P3.3) * CF_{P3.3} + EW_{T_n}(P3.2) * CF_{P3.2} + \\ EW_{T_n}(P3.5) * CF_{P3.5} + EW_{T_n}(P3.1) * CF_{P3.1}.$$

The terms used in the formula can be defined as follows:

$E(T_n)$: relative effectiveness, for technique T_n ;
 $EW_{T_n}(P3.X)$: effectiveness weight for technique T_n with respect to problem 3.X;
 $CF_{P3.X}$: criticality factor for problem 3.X.

Figure A.2-1 illustrates the results of applying the effectiveness formula to all techniques. The techniques are shown as a prioritized list in order of decreasing effectiveness.

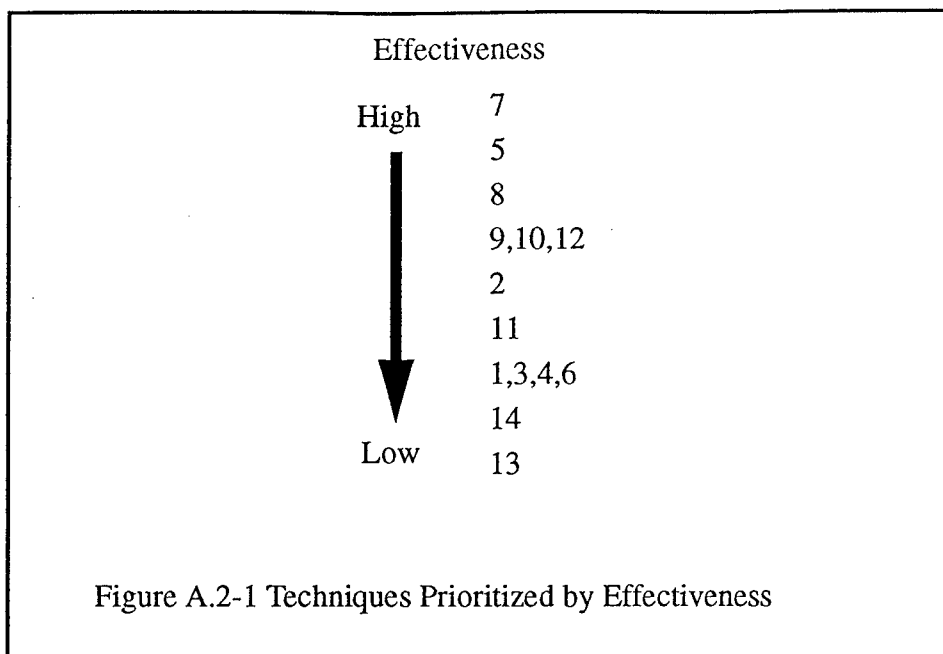


Figure A.2-2 below illustrates the ranking of techniques according to their potential for automation. The techniques are arranged from high automation potential to low automation potential.

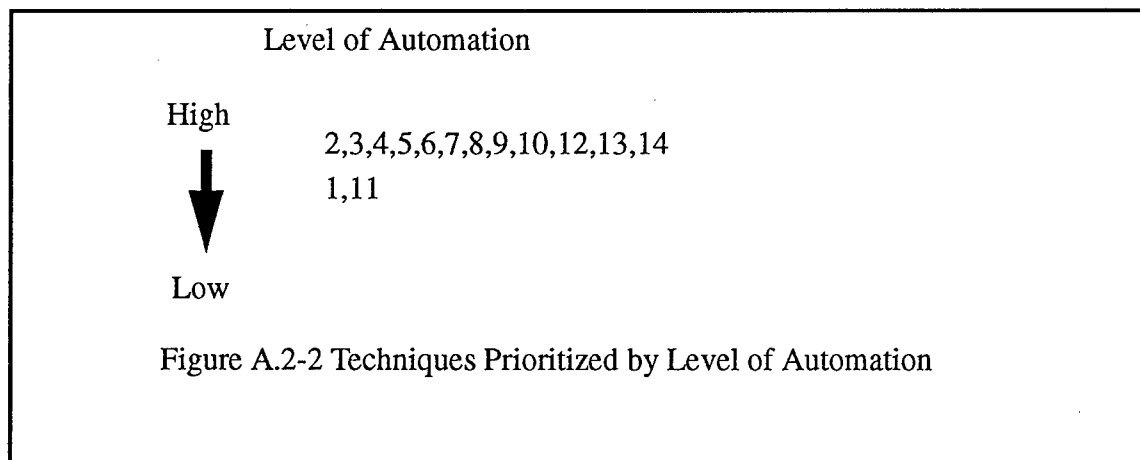


Figure A.2-3 illustrates the combination of effectiveness and automation rankings. The overriding goal of this combined ranking is to guide the application of software development resources for both near-term and far-term parallel software testing tools.

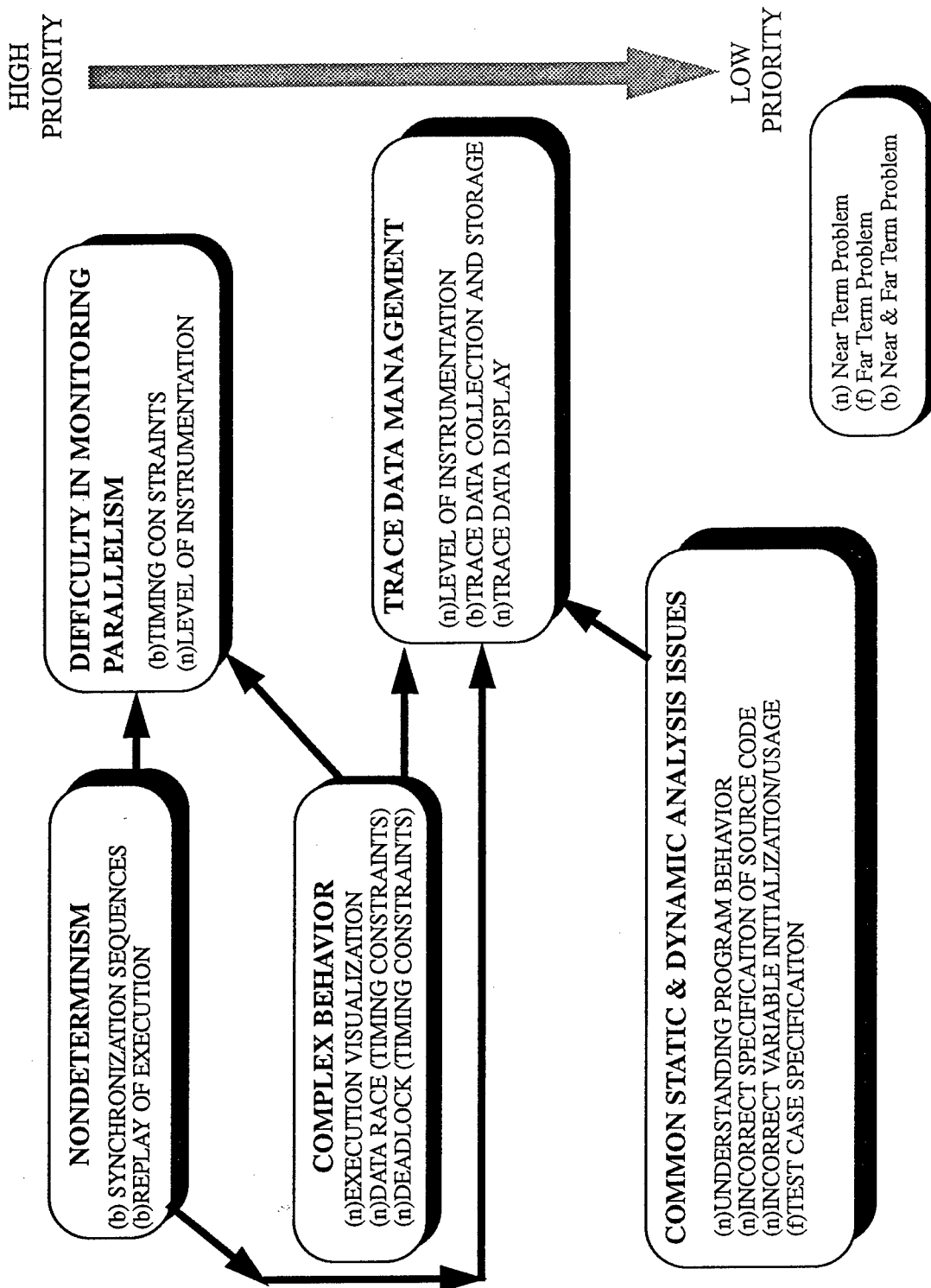


Figure A.2-3 Combined Prioritization of Techniques

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.